HALCON Version 6.0

HALCON/ COM

User's Manual

How to use the image analysis tool HALCON, Version 6.0, in your own COM programs

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher.

1. Edition April 1999
2. Edition October 2000

Copyright © 1999-2000 by MVTec Software GmbH, München, Germany

Microsoft, Windows, Windows 95, Windows NT, Windows 2000, Visual C++ , and Visual Basic are either trademarks or registered trademarks of Microsoft Corporation.

All other nationally and internationally recognized trademarks and tradenames are hereby recognized.

More information about HALCON can be found at:

`http://www.mvtec.com/halcon/`

About This Manual

This manual gives an overview of the HALCON/COM interface. COM ("***c**omponent **o**bject **m**odel*") is a Microsoft standard for component-based, language-independent software development. This manual contains all necessary information to install the COM interface and understand its main concepts; a guided example shows how to develop HALCON applications using Visual Basic.

This manual is intended for the advanced HALCON user. You should be familiar with the main HALCON package and have experience in developing image analysis applications with HDevelop. Of course you must have knowledge about the language/tool you plan to develop with (e.g. Visual Basic).

This manual is divided into the following parts:

- **Introduction**
  This chapter provides a short overview of COM concepts in general.
- **The HALCON/COM Interface**
  This chapter describes the structure of the HALCON/COM interface giving a quick overview, though not getting very deep into the details.
- **Example Session**
  In this chapter a small image processing demo application is developed stepwise with Visual Basic.

Release Notes

Please note the latest updates of this manual:

- 2^{nd} **Edition, HALCON 6.0 (October 2000)**
  Revision of the structure of the manual and minor updates.

Contents

1 Introduction **1**
- 1.1 Why Components? . . . 1
  - 1.1.1 Interfaces and COM . . . 2
  - 1.1.2 Objects, Methods and Data Members . . . 2
  - 1.1.3 Inheritance and Polymorphism . . . 2
  - 1.1.4 Early and Late Binding . . . 3
- 1.2 HALCON and COM . . . 3
  - 1.2.1 Advantages . . . 3
  - 1.2.2 Disadvantages . . . 3
  - 1.2.3 Limitations . . . 4
- 1.3 Additional Sources of Information . . . 4

2 The HALCON / COM Interface **5**
- 2.1 Preface . . . 5
- 2.2 More about Classes . . . 6
  - 2.2.1 Different Types of Classes . . . 6
  - 2.2.2 Classes for Special Purposes . . . 7
- 2.3 Object Construction and Destruction . . . 8
  - 2.3.1 Construction . . . 8
  - 2.3.2 Destruction . . . 9
- 2.4 Interfaces and Inheritance . . . 11
- 2.5 Methods and Properties . . . 11
- 2.6 A closer Look at Data Types . . . 12
- 2.7 Error Handling . . . 12
- 2.8 HALCON/COM and Visual Basic . . . 13
  - 2.8.1 Object Instantiation . . . 13
  - 2.8.2 Error Handling . . . 14

3 Example Visual Basic Session **17**
- 3.1 First Step: The GUI . . . 17
- 3.2 Second Step: Functionality . . . 20
- 3.3 Final Step: More Functionality . . . 21
- 3.4 Other Examples . . . 22

Bibliography **25**

Index **27**

Chapter 1

Introduction

This chapter provides a short overview of the main aspects of component-based software development and shows how this can be done with COM. The advantages and disadvantages of component-based software engineering are discussed as well as the limitations as far as HALCON is concerned.

It is not intended to discuss all aspects of COM in detail in this manual. Thus anyone who wants to know further details about programming with COM is recommended to read the corresponding literature.

1.1 Why Components?

Component-based software engineering has been discussed intensively in the past few years. Concrete standards evolving from this discussion include DCE, CORBA and COM. The fascination about components is their ability to be reused at binary level. With their slight (not always very complete) object oriented features and the option for location transparency they form a good base for the development of modern, distributed software systems. In fact, COM *is* the base for nearly every software technology Microsoft is dealing with: ActiveX, OLE and DirectX for example base on COM, thus large parts of the Microsoft operating systems are component-based. Let's have a look on what a COM component looks like. The main features of COM components are:

1. Reuseability at *binary* level
2. Object orientation
3. Robust versioning
4. Location Transparency

In fact, the first topic is the most important in our case. Binary reuseability implies that an existing component can be used within virtually any software development tool in any language, as long as there is a certain basic support for COM in it. Furthermore, it does not matter what tool was used to implement the component as it is a binary standard. There are numerous different tools for developing COM components, e.g. MS Visual C++, Borland C++ and Delphi, MS

Visual Basic, MS J++, and a lot more. The object oriented characteristics of COM components such as encapsulation, inheritance and polymorphism ease the development of software especially for those programmers that are already familiar with object oriented programming (e.g. with C++). The versioning and location transparency features of COM components will not be discussed any further here, as they are less important for developing software with HALCON and as they are not yet completely tested and implemented with the current revesion.

1.1.1 Interfaces and COM

An important aspect of COM is the usage of component interfaces. Compared to the C++ terminology an interface corresponds to the *declaration* of a class, thus showing its functionality and hiding the (internal) implementation details. This encapsulating feature makes it possible to efficiently use a COM component without the need to know anything about its implementation. However, unlike a C++ class a COM component can have *multiple* interfaces. Seeming rather strange at first glance, the motivation for this gets clearer when looking at the versioning aspect: once defined, an interface stays exactly the way it is; newer versions of the component then define enhanced or extended functionality by new interfaces allowing existing software to remain unchanged, because it can still use the older interfaces. Another reason for multiple interfaces is the ability to partition a component's functionality into several well-defined portions. There are also a couple of standard COM interfaces every component has by default; those are mainly `IUnknown` and `IDispatch`. They are not described in this manual so that interested readers are refered to the appropriate literature.

1.1.2 Objects, Methods and Data Members

Keeping the C++ point of view we come now to the question: what are the objects, methods and data members of a class in terms of COM? Just as with C++ classes there are interfaces in COM and those interfaces are made up from some methods. Several objects of a class correspond to several instances of a component, each with its own internal state. However, there are some differences with data members, as their handling within COM depends on the tool the component is used by. COM follows a concept of *properties* that are special methods allowing tools like Visual Basic to treat them just like data members. With C++ they still are only methods, thus there is no way to modify and retrieve data members directly. From this point of view COM class data members can be compared with C++ private class members.

1.1.3 Inheritance and Polymorphism

As COM is a binary standard, inheritance must also take place at binary level which makes the process slightly uncomfortable. Without going into detail one can say that the only thing of interest here is that there are two methods of reusing existent components: *containment* and *aggregation*. Both techniques are commonly used by C++ programmers as well: containment corresponds to C++ classes instantiated as a member object in other classes, whereas aggregation roughly corresponds to inheritance. The main difference between containment and aggregation is the way the interface(s) of the contained/aggregated component(s) are handled: the methods and properties of a *contained* component are hidden to the outside so that only the containing component can use them. Any method that should be visible from outside has to be

re-defined in the outer component. In contrast to this, the interface(s) of an *aggregated* component are merged with the interfaces of the aggregating one thus automatically making their methods visible to the outside. The object oriented feature of polymorphism is also achieved through interfaces: different COM classes exposing the same interface can be understood as showing polymorphic behaviour, as they act differently responding to the same methods.

1.1.4 Early and Late Binding

Binding is the process of resolving the call adresses of a component's exposed methods. This can be done at compilation time (*early binding*) or at runtime (*late binding*). Thus when using a component a programmer can take a specification of the contained methods and integrate it directly into the application. Therefore, the specification must be available in a format that fits the used programming language (e.g. when using C++ as client language one would typically include the header-files containing the method declarations). Otherwise the programmer could make the methods being bound at runtime — then no language-dependent information is needed for compiling and the calling mechanism is totally different. Again we will not go into details and thus only point out the main aspect that runtime-bound methods calls are somewhat *slower* than their early-bound counterparts. The HALCON/COM interface supports both early and late binding so that special needs of different applications can be satisfied.

1.2 HALCON and COM

1.2.1 Advantages

How can the user of HALCON profit from a COM interface? The main advantage is that with a COM interface the powerfull image processing features of HALCON can be used by a wide range of applications and development tools so that its usage is no longer restricted to C and C++. The world of Visual Basic, Delphi and other tools stands open for the development of HALCON applications. Beside this, another important advantage of a COM interface lies in the widespread standard HALCON is joining with it: nearly any piece of software related to Windows is also related to COM. This will not change in the near future and upcoming new development tools will be firmly bound to COM anyway. Further versions of the HALCON/COM interface might also support COM's ability to enable multi-client, multi-threaded application development with method calls across process and machine boundaries.

1.2.2 Disadvantages

Of course COM can also show some disadvantages and using HALCON in a component-based way can be rather tricky depending on the language and tool one is using for development. For example it is really easy to use software components with Visual Basic, but this is not necessarily the case when considering VisualC++ — here the "classic" HALCON/C++ interface is far more convenient. As said before, COM shows some other powerful features like options for distributed computing that may be supported in future HALCON releases. Then the HALCON/COM interface should be a big step ahead the other language interfaces.

1.2.3 Limitations

Although the complete functionality of the HALCON library can be used without limitations, there exist some limitations for the current version of the HALCON/COM interface:

- Parallel HALCON has no language interface to COM, because Microsoft Visual Basic, the main development tool for COM programs, does not support real multi-threading. If you need the features of Parallel HALCON (automatic parallelization or reentrancy) please use the C or the C++ interface.
- Currently, the HALCON Extension Package Interface cannot be used to create extension packages for COM.

1.3 Additional Sources of Information

For further information you may consult the following manuals:

- **Getting Started with HALCON**
  An introduction to HALCON in general, including how to install and configure HALCON.
- **HDevelop User's Manual**
  An introduction to the graphical development environment of the HALCON system.
- **HALCON/C++ User's Manual**
  How to use the HALCON library in your C++ programs.
- **HALCON/C User's Manual**
  How to use the HALCON library in your C programs.
- **Extension Package Programmer's Manual**
  How to extend the HALCON system with your own operators.
- **Frame Grabber Integration Programmer's Manual**
  A guide on how to integrate a new frame grabber in the HALCON system. Note that in some cases you might define new operators (using the Extension Package Interface) instead of using the standard HALCON Frame Grabber Integration Interface in order to exploit specific hardware features of a frame grabber board.
- **HALCON/HDevelop, HALCON/C++, HALCON/C, HALCON/COM**
  The reference manuals for all HALCON operators (versions for HDevelop, C++, C, and COM).

All these manuals are available as PDF documents. The reference manuals are available as HTML documents as well. For the latest version of the manuals please check

```
http://www.mvtec.com/halcon/
```

Chapter 2

The HALCON COM Interface

This chapter describes the structure of the HALCON/COM interface. Please note, that there are several interesting facts to know about COM that are not discussed here, because they are not directly relevant for working with the HALCON/COM interface. Thus, you are recommended to read additional COM-related literature and/or the documentation of your preferred development tool (e.g. Visual Basic). At least you should be familiar with object oriented concepts like inheritance, classes and methods in order to understand the following chapter.

2.1 Preface

Let's have a closer look at the HALCON/COM interface. The whole package (called the *HalconX library*) contains all the functionality partitioned into several *classes*. Each class has one or more *interfaces* that are built up from *methods* and *properties*. The HALCON/COM interface uses inheritance and thus contains derived classes and also an abstract base class. Since COM is not meant to supply a standardized inheritance mechanism, HALCON/COM makes extensive use of interfaces to simulate inheritance (we will discuss this topic more detailled afterwards).

Naming Conventions:

- ▷ classes are capitalized and begin with an "*H*". They always end with an upper-case "*X*"; for example: `HFramegrabberX`.
- ▷ interfaces are also capitalized and end with "*X*", but begin with an "*I*"; for example: `IHObjectX`.
- ▷ methods, properties and parameters are capitalized; for example: `GrabImage`, `FileName`.

Since COM is only available for the Windows operating systems family, all file paths and environment variables in this manual are printed in the Windows NT convention, e.g.,

```
%HALCONROOT%\examples\vb\Manual
```

to denote a subdirectory containing an example package within the HALCON base directory referenced by the environment variable `HALCONROOT`.

2.2 More about Classes

Classes are the essential items to deal with when writing HALCON/COM applications. There are quite a lot of them and there are different types of classes, thus we will have a closer look on how classes are used and what attributes they have. The classes of the HALCON/COM interface are partly related in a way which could be described as inheritance. Therefore it is important to get an impression of how inheritance is achieved within HALCON/COM and how to use it.

2.2.1 Different Types of Classes

There are two major categories of classes:

1. classes instantiating objects that have an internal state and
2. classes instantiating objects with no internal state.

The first category is used to implement special data structures (like images, files, frame grabbers, etc.) and the operators belonging to this data, whereas the second category is only used to group operators belonging together. If several objects of a class belonging to the first category are instantiated, they are all different from the HALCON point of view, whereas the second category does not have this quality. For example if we consider several objects instantiated from the image class `HImageX`, they all denote something different: they represent different HALCON images. However, if we have an object of a class like `HSystemX` that represents the internal state of the HALCON kernel, it does not matter how many of those objects will be instantiated, because all of them denote the same (there is only one HALCON kernel to be represented). Those classes can be understood as *group classes*, denoting that they supply a bunch of methods that all have some semantic peculiarities in common. In contrast to this, methods of the first category classes may share a common semantics, but work on different data. For example if an object of the `HImageX` class is instantiated, its methods always work exactly on one specific HALCON image: the one that is represented by the object (to be precise a `HImageX` object may also represent an *array* of images). Different `HImageX` objects represent different images and therefore work on different data. From an object oriented point of view classes and data types are the same so that all of the HALCON/COM classes can be seen as data types. To make a long story short, it does not make much sense to see the second category classes as data types, because they have no “meaning”. That’s why they will never show up as a method parameter.

Beseides the first two groups, we can categorize classes in another way well known in the object oriented world:

1. abstract classes and
2. non-abstract classes.

A class is called abstract, if it can not be instantiated. Thus an abstract class must be the base class for other classes, if it should be of any use. The idea behind this concept is that abstract classes provide a semantic base for their derived classes without being concrete enough for

instantiation. A look on real world terms reveals many analogous cases: there is for example the class *animal* which can be seen as an abstract base class for classes like *fish*, *bird* and *horse*. Any existing being can not be only an animal, it is always either a fish, a bird or a horse (or whatever else). There is only one such abstract class in the HALCON/COM interface: `HObjectX`. It represents a HALCON iconic object, such as an image, a region or an XLD. The derived classes (`HImageX`, `HRegionX` and so on) then specify the exact type of the HALCON iconic object (see also the class overview in figure 2.1).

2.2.2 Classes for Special Purposes

There are some HALCON/COM classes that have special jobs. Although they do fit into the above systematics, they are worth mentioning, because they have to be used in a specific way or show some specific semantics:

2.2.2.1 HWindowXCtrl

This class is a HALCON window in the form of an *ActiveX control*. Its advantage against the ordinary HALCON window (`HWindowX`) is the possibility to exist inside an *ActiveX container*. An ActiveX container is a part of a window which can contain GUI elements, such as buttons, sliders and so on. When developing complex image processing applications, it is often necessary to integrate all user interaction components (input and output) seamlessly into one surface. The `HWindowX`-type HALCON window does not provide this flexibility, since it always appears as a top-level window. In order to behave like an ordinary `HWindowX` HALCON window `HWindowXCtrl` uses the COM technique of *aggregation*. Thus any new created `HWindowXCtrl`-type ActiveX control automatically instantiates a `HWindowX` object which is bound to the control.

2.2.2.2 HUntypedObjectX

`HUntypedObjectX` is derived from `HObjectX` just like `HImageX`, `HRegionX` and so on. Its purpose is to get an instantiable form of the abstract base class. The class does not have any members, as it just consists of a polymorphic data type without any special meaning. As it is this weak typed, it can not be used together with the strong typed methods of the other classes, except by using the `Cast()` method which allows arbitrary type conversions between all `HObjectX` derived classes (explained later on). `HUntypedObjectX` is meant to be a generic data type used by the special class `HOperatorSetX`.

2.2.2.3 HOperatorSetX

This is basically a group class for all existing HALCON operators. `HOperatorSetX` is meant to provide access to a procedural way of HALCON programming. The reason for that is the fact that it is easier for some non object oriented tools like HDevelop to generate COM code automatically when using a procedural technique. The specificity about `HOperatorSetX` is that all its methods exclusively use `HObjectX`-type input iconic parameters and return `HUntypedObjectX`-type output iconic objects. **Note**: it is allowed to have input parameters

of an abstract type! C++ programmers will know this concept — it corresponds to passing objects through a *pointer* to their abstract base class. However, this is not allowed for output parameters. Here the data type must be specified, because there has to be some (typed) variable which can store the output object. Since `HOperatorSetX` methods must use a generic data type scheme, output iconic objects are always of type `HUntypedObjectX`. Hand-written COM code should not make use of the `HOperatorSetX`/`HUntypedObjectX` concept, since it weakens the otherwise strong typed object oriented approach; it becomes relevant only in cases, where automatically generated code is involved.

2.2.2.4 The Method `Cast()`

All classes derived from `HObjectX` supply the `Cast()` method which is used for type conversions between the concerned classes. This method breaks up the well defined data type scheme evolving from the class hierarchy, so it should not be used except when there is no other possibility! By using the `Cast()` method an image object can be turned into a region, an XLD can become a `HUntypedObjectX` object and so on. The reason for that is, as mentioned above, the need to convert back and forth to the `HUntypedObjectX` objects used by the `HOperatorSetX` methods. For example if an automatically generated code fraction produces an object variable of type `HUntypedObjectX` which shall be used in some hand written code as an image, it can be forced to become a `HImageX` object. Of course, it must be assured that the variable really *does* contain an image (it is the programmer's task to take care for that). We will have a closer look on how to use the `Cast()` method later on.

2.3 Object Construction and Destruction

2.3.1 Construction

A HALCON/COM object can be in different states. It can be *instantiated* and it can be *initialized*. An instantiated object need not necessarily be initialized, but an initialized object is always instantiated. Instantiation means using an appropriate technique to produce a "living" COM object (this techniqe differs according to what client language is used). Initializing means giving the object a well-defined internal state. There is another state an object can have: it is neither instantiated nor initialized. Since not being instantiated means not existing, what exactly does this condition mean? The answer depends somewhat on the client language used, but is rather easy to understand, if we realize that COM object variables actually are *references* to an existing (or not existing) object. Thus if such a variable is declared, there is not necessarily yet an object created it refers to. As we like to mix up the terms "reference to an object" and "object" in this manual, we may speak of an uninstantiated object in that case. For example if an `HImageX` object is created (not only the referring variable is declared), it is useable in terms of COM (since it is instantiated), but it does *not* yet contain any HALCON image. To be a valid HALCON/COM object, it still must be initialized. This can be done in two different ways:

- the object can initialize itself or
- the object can be initialized by another object's method.

In the first case a so-called *constructor* is executed. The term "constructor" is somewhat misleading here, since it has a slightly different meaning than for example a C++ constructor. In C++ terms a constructor is a special method (always named exactly as the class) which does the object construction *automatically* without the need to be called explicitly. In the HALCON/COM case a constructor is an ordinary method which initializes the object's internal state and must be called explicitly. For that reason a HALCON/COM class can have many (differently named) constructors. One of the constructors of our example `HImageX` class is `ReadImage()` which initializes the object by reading an image file from the hard disk. Another way to initialize an object is to get it as result after calling another object's method. For example an uninitialized `HImageX` object becomes initialized, when it is returned from a call to `MeanImage()` (which does a convolution on an image and produces another image).
Note: the mechanism is indeed slightly more complicated and again depends on the client language. In fact, an object variable needs *not* refer to a living COM object, when being used as a return value of a method call. Even more, if it *does*, the referred object gets destroyed before the new object is assigned to the variable.
Actually, there is yet another way to initialize a HALCON/COM object: by using the `Cast()` method mentioned before. The cast method takes another object of a related (derived from `HObjectX`) class as parameter, which in turn has to be initialized. Then the object the calling method belongs to gets initialized with the internal state of the other object. Because no copying or duplication takes place, the other object gets *uninitialized*.

Note: Not all HALCON/COM objects can have this initialized state! An object can only be initialized, if it *has* an internal state. Thus all the group classes from 2.2.1 must be excluded. Therefore they have no constructor methods.

2.3.2 Destruction

We have seen that creating a valid HALCON/COM object is a two-step process: first the COM object needs to be instantiated, then it must be initialized. This implies also a two-step destruction process: to destruct an object it must be uninitialized before being destroyed (which is the opposite of being instantiated). The latter is done by COM and the client environment, the first must be performed by some *automatic destruction process*. To pick up the `HImageX` example again, the automatic destructor must free the image data using the HALCON memory management. Another example: if a top level HALCON window gets initialized, it is physically opened. The automatic destructor must take care of the window getting closed before the according COM object finally gets destroyed.

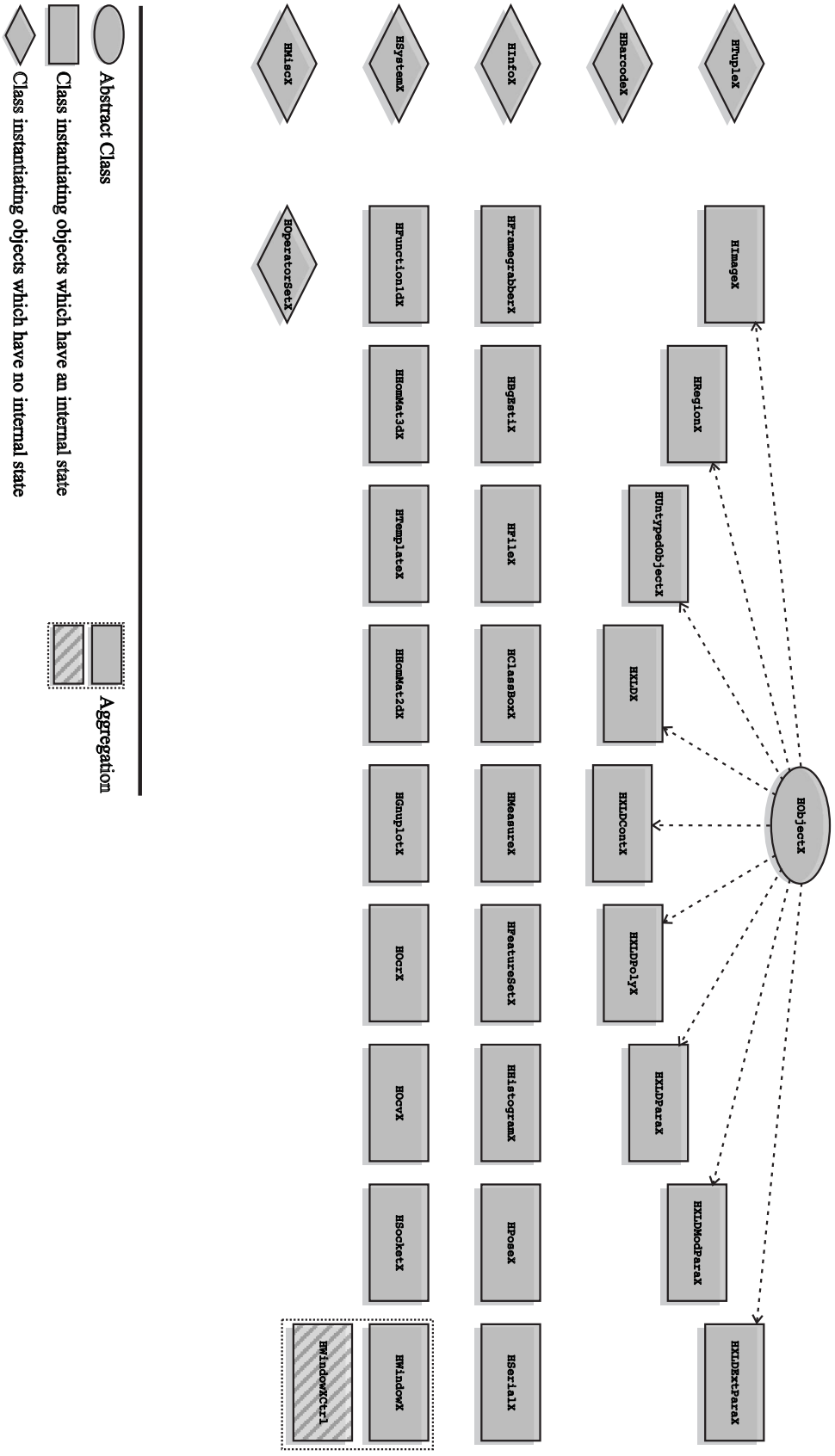

Figure 2.1: *An overview of the HALCON/COM class hierarchy.*

2.4 Interfaces and Inheritance

As said before, COM classes are built up from *interfaces* which in turn contain methods and properties. Each HALCON/COM class has got one default interface named accordingly to the class. For example the class `HImageX` contains the default interface `IHimageX` which provides all the relevant methods. Dealing with interfaces is strongly related to the client environment used, so we will not have a too close look at this topic now. For example Visual Basic tries to completly hide the whole interface topic from the user — in that case, an interface and the related class are the same most of the time. Actually, an interface behaves similar to a C++ pointer or reference to an object and that is exactly what Visual Basic tries to hide. As said before, interfaces are used for inheritance simulation within HALCON/COM; the way this works is simple: We have seen, that `HObjectX` is an abstract (and thus not instantiable) class. On the other hand, the derived classes `HImageX`, `HRegionX`, etc. have to supply their own functionality *plus* the inherited `HObjectX` functionality. This is done with the help of interfaces: currently there is *no* COM class named `HObjectX` (thus no `HObjectX` object can be initialized), only an interface named `IHObjectX`. This interface appears in all “derived” classes together with their default interfaces. `HImageX` for example has two interfaces (in fact it has a hidden third one):

1. `IHImageX` (the default interface) and
2. `IHObjectX` (the inherited interface).

This allows to satisfy every method which expects a `HObjectX` parameter (actually it expects a reference to that class in form of the `IHObjectX` interface) with any derived class object, as such an object always has the `IHObjectX` interface, too. This corresponds to a certain object oriented rule which allows an automatic cast from the derived class to the base class (not the other way). How intuitive this feature can be used again depends on the client language/tool: Visual Basic 5.0, for example, only regards the default interface an object supplies and treats that as if it was the class itself. That implies that only the methods contained in the default interface seem to belong to a class. Thus to also use the “inherited” methods (contained in the `IHObjectX` interface) an explicit cast to the base class is necessary. This is a widely known Visual Basic weakness and may get improved by Microsoft in forthcoming versions.

2.5 Methods and Properties

There is not much to say about methods, since they can be used quite intuitively. The only interesting aspect here is the fact that different classes can have methods with the same name, but with a different parameter configuration. These methods perform the same action (as their identical names let expect), but “show a different point of view”. For example the method `GrabImage()` that retrieves an image from a frame grabber is part of the `HImageX` as well as of the `HFramegrabberX` class. In the first case, the method is a constructor for an image object and takes a `HFramegrabberX` object as parameter (that denotes the frame grabber from which to retrieve the data). In the second case, the method takes the `HImageX` object into which the image should be grabbed as parameter.

Properties are a special COM feature and can be treated like data members by tools like Visual Basic. There are two kinds of them: *put* and *get* properties. A put property allows the user

to change the internal state of an object, whereas the get property only allows to read out the internal state. Usually (but not always) both types are present acting like an ordinary variable class member in C++. The properties in the HALCON/COM interface are just for convenience purposes, since they always map to an existing method. The `HWindowX` class for example has quite a lot of properties: `Width` and `Height` define the window's extent, `Draw` sets or gets the current drawing mode, and so on. All of these properties are mapped to their corresponding methods. Reading out the draw mode for example results in a call to the Method `GetDraw()`, whereas `Width`/`Height` both are mapped to the method `GetWindowExtent()` (when retrieving values) and `SetWindowExtent()` (for changing values, respectively).

2.6 A closer Look at Data Types

We have seen that a lot of the data types used within the HALCON/COM interface are actually the classes themselves. But there are also more basic, "everyday" types being used. A widely used data type in the COM world (and thus in Visual Basic) is the `VARIANT` type. All users of the HALCON/C++ interface will know the `HTuple` data type which is both polymorphic (it can be one of several different "subtypes") and supplies array functionality (it can hold several data items at once). `VARIANT`s behave analogously and are the COM equivalent for `HTuples`. Exactly like a `HTuple` a `VARIANT` is able to hold data of different basic types plus the information what kind of data it contains. Also a combination of different or equal values is possible just like with `HTuples`. The main difference is that a `VARIANT` is no class. This implies the complete absence of any methods. So all the additional functionality of the very powerful `HTuple` class must be accessed in another way. For that reason there is a `HTupleX` class which groups methods for tuple operations like vector addition, string concatenation, and so on. Another important and widely used COM data type is `BSTR`. This standard COM flavor of character strings is not directly compatible with standard C-like string implementations, mainly because it uses wide chars. This means, that auxiliary functions must be used to access or modify `BSTR`s when using C/C++. Again this means no problem with Visual Basic, as it is the default string data type there. Additionally, there are integral data types like `long` and `double` as well. They are used in situations where array- or multitype-parameters are not allowed or make no sense.

2.7 Error Handling

The HALCON/COM interface uses the standard COM error handling technique where every method call passes both a numerical and a textual representation of the error to the calling framework. It is then up to the caller to react to this information. Since low error numbers are reserved for COM, HALCON/COM uses a very high offset to its own error codes. To get the correct HALCON error code this offset must be subtracted from the received code. The offset is a (read only) property of the `HSystemX` class. There are two offsets: one for the HALCON/COM *interface* and one for HALCON itself. Those properties are named:

- `HSystemX.ErrorBaseCOM` and
- `HSystemX.ErrorBaseHalcon`.

In order to get the correct HALCON error code a `HSystemX` object must be instantiated (one is enough for the whole application anyway, since `HSystemX` objects have no "identity"). Then the value of the respective `HSystemX`-property must be subtracted from the returned error code to get the correct HALCON error code. For an example how to deal with error codes, see section 2.8.2.

2.8 HALCON/COM and Visual Basic

So far the important basics of the HALCON/COM interface have been explained. Now let's have a look at what things behave like when using Visual Basic .

2.8.1 Object Instantiation

There are many different ways to instantiate COM objects in Visual Basic. We will discuss only one of them, because it has certain advantages over all the others. We have seen in the sections before that a distinction should be made between the instantiation and the initialization of objects. Even more important we should also distinguish objects from object reference variables. An object reference variable is set up by its declaration with the keyword `Dim`:

```
Dim image1 As HImageX
```

This statement does *not* yet create a COM object, it just declares a variable able to reference a `HImageX` object. If we want to declare a reference variable and immediately create an object it refers to, we should write

```
Dim image1 As New HImageX
```

Now a "new" `HImageX` object is created and the variable '`image`' refers it.
Note: variable declaration is not a must with Visual Basic, but should always be done anyway! Undeclared variables get declared automatically when referenced and that can lead to errors which are very hard to discover! It is a good idea to place the statement '`Option Explicit`' on top of every Visual Basic module, because then variable declaration is forced.
We now have a valid COM object, to which the declared variable refers. To initialize this object, we could call a constructor method:

```
Dim image1 As New HImageX
Call image1.ReadImage('some_file_name')
```

Note the `Call` keyword! It's necessary in Visual Basic, if the called method doesn't return a value. The other way of initialization would be using another object's method:

```
Dim image1 As New HImageX
Dim region1 As HRegionX
Call image1.ReadImage('some_file_name')
Set region1 = image1.Threshold(128, 255)
```

There are two important things here. First, the `Set` keyword: it replaces the `Call` keyword when the called method returns another COM object (in this case a region). Furthermore, the second variable declaration omits the `New` keyword. This is because the corresponding variable need not instantiate an object at declaration time. This is done within the `Threshold()` method, which creates a new COM object itself and passes a reference to this object as its return value.

HALCON/COM objects get destroyed as soon as no variable references them anymore. For local variables, this is the case when they go out of scope (e.g. a subroutine is left). For global variables, or if an explicit destruction is desired, this has to be done by the user:

```
Dim image1 As New HImageX
Dim region1 As HRegionX
Call image1.ReadImage('some_file_name')
Set region1 = image1.Threshold(128, 255)
Set image1 = Nothing
Set region1 = Nothing
```

Here, both variables are assigned the special Visual Basic keyword '`Nothing`' which denotes that they do not reference their related COM objects anymore. These COM objects thus are not referenced at all which leads to their immediate destruction.

There is, of course, a lot more to say about Visual Basic/HALCON programming. Some further aspects might become clear in the following example session.

2.8.2 Error Handling

When using Visual Basic, errors can be trapped by an error handler. If no custom error handler is present, Visual Basic itself supplies a default one, which shows a message box containing the textual error description. Error handlers in Visual Basic can be set with the keyword `On Error`. To trap an error in a portion of code, the apropriate construct could look like this:

```
Dim LastErrorCode As Long
Dim SysObject As New HSystemX

On Error Goto myErrorHandler

<some code>

myErrorHandler:
  ' do something with the error information, for example:
  Debug.Print "Error occurred: " + Err.Description
  LastErrorCode = Err.Number - SysObject.ErrorBaseHalcon
  Resume Next
```

If an error occurs in `<some code>`, an immediate jump to the label `myErrorHandler` is made, where an arbitrary error processing mechanism can be placed. The scheme used in the example tries to model a traditional, 'procedural' error recovery strategy, where every function call returns an error code, which has to be checked before program execution can continue. When an error occurs, the error handling code at the corresponding label takes over, writes a status

message (the textual error representation) to the Visual Basic'Immediate'-window and stores the error code in a global integer variable. The global Visual Basic object `Err` is the source of information in this case. Afterwards, control is put back to the line following the statement which produced the error. This is done by the Visual Basic command `Resume Next`. The next line of code then would be responsible for checking the error code stored in `LastErrorCode`. Of course, this method is a little long winded and not quite up-to-date — an exception handling mechanism in general is much more flexible and robust — but it's quite easy to understand. We have seen, that there are two types of errors: HALCON-related errors and COM-interface-related ones. Since the COM-interface errors have smaller error numbers than the HALCON error codes, the above mechanism would lead to negative numbers. In that case, the produced error code would have to be subtracted from `SysObject.ErrorBaseCOM` to get the correct (then interface related) error code.

Chapter 3

Example Visual Basic Session

In this chapter you will learn how to develop HALCON applications quickly using Microsoft Visual Basic and the HALCON/COM interface. There will be simple steps describing what to do. The result will be a very small exemplary vision application with doubtful value but equipped with a graphical user interface. As an additional source of information you are strongly encouraged to have a look at the other examples which are supplied as Visual Basic sources together with HALCON.

The program developed in this chapter is also available ready to go in the `%HALCONROOT%\examples\vb\Manual` directory together with other examples. However, it is recommended to follow the steps below and program it yourself, as you will get a better impression of how Visual Basic program development works and gain a lot of additional information with the single steps.

Please note, that to use HALCON/COM inside Visual Basic you need Windows NT 4.0 with Service Pack 4 or Windows 2000, and Visual Basic 6.0.

3.1 First Step: The GUI

Go ahead and

1. Launch Visual Basic. A dialog named `New Project` should appear allowing you to select the type of project you want. Switch to `New` in the tab list, select `Standard EXE` and click `Open`.

2. Select `Project` from the menu bar and click `Components`. A dialogue box shows up listing the components installed on your system. Switch to `Controls` in the tab list and place a check next to the item `HalconX typelibrary V1.0`.

3. Press `F2`. The object browser should appear. See if you can find `HImageX` and browse through some of the corresponding methods. Clicking on a method shows its parameterization as well as a short help text about what it will do in the status area at the bottom of the object browser. Close the object browser.

4. Have a look at the dialogue template (’*form*’) showing in the lower half of the screen; it should be titled `Form1`. In the upper half you will discover an area titled `Properties - Form1`. Here you can set and retrieve the active GUI object’s (in this

case the form's) properties. Click on `Form1` right beside `Caption` and change the string to `HalconX example`. You should see the effect of your action immediately in the caption text of the below form.

5. Grab the form and resize it to a suitable extent.

6. Have a look at the tool bar to the left: here you can find all the control elements you can place inside your form. They are represented as small icons. Move the mouse cursor over the different icons to see the bubble help revealing their names. You should find an icon showing the HALCON symbol named `HWindowXCtrl`. You guessed it! That is our ActiveX control HALCON window.

7. Activate the `HWindowXCtrl` icon. Draw a rectangular region inside the form — make sure it is approximately square. When releasing the mouse button the square area should turn black.

8. Switch to the `CommandButton` icon (looking like a brick) in the left tool bar. Draw a button inside the form beside or below the HALCON window. Change the button's caption text to `Next >>` in the properties box.

9. Now switch to `Label` in the tool bar and draw a longish rectangular area at the bottom of the form. If you encounter placement difficulties due to lack of form space, you can always resize the form to fit your needs.

10. Resize the form so that it fits around the before created items. Now you have the entire GUI for your application ready to go and your screen should look similar to figure 3.1.

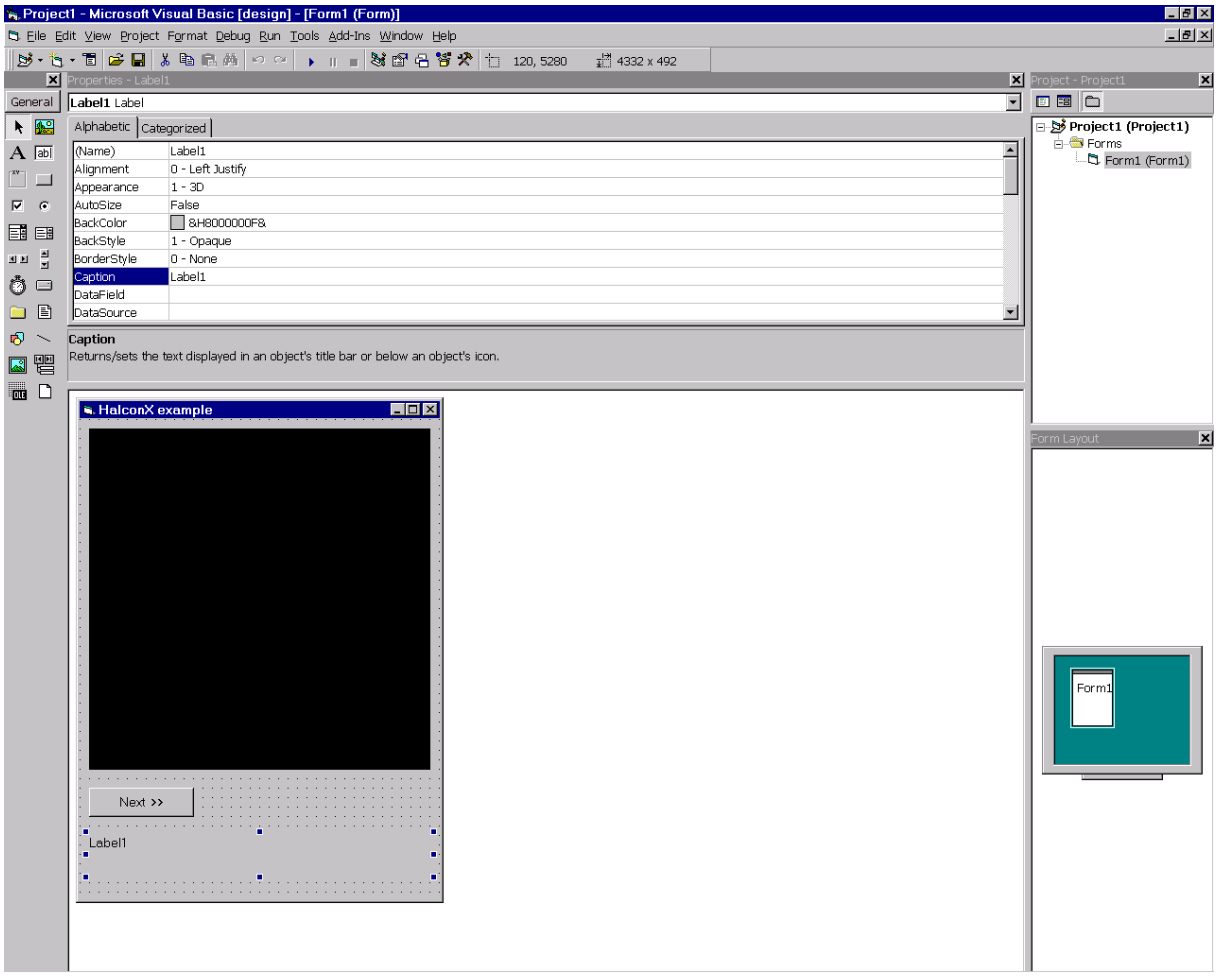

Figure 3.1: *Having performed all the steps from 3.1 you should end up with a setup like this.*

3.2 Second Step: Functionality

Now you have the GUI finished, you should go ahead and make the application do something:

1. Right-click somewhere inside the form and select `View Code`. Another window will pop up over the form with two combo boxes at its top border. Select `Form` from the left combo box. You will see the code to be executed when the form is created.

2. Insert a line into the subroutine:

```
Private Sub Form_Load()
    Label1.Caption = "Click Next to start"
End Sub
```

   You just changed the text the label at the bottom will show when the application is launched.

3. Next we will declare some important variables: switch back to `(General)` in the left combo box above the source code window and insert in the following lines at the top:

```
Dim Monkey As New HImageX
Dim Window As HWindowX
```

   Some online selection boxes for the desired object type will assist you. We have just created two objects: a `HImageX` and a `HWindowX`. The reason for the `New` keyword in the first line is that we want the `HImageX` object to be instantiated (e.g. memory being allocated for it). This is not necessary for the `HWindowX`, since it is already instantiated — it is the ActiveX control we have drawn inside the form.

4. The object '`Monkey`' is instantiated as we know (although it is not yet initialized with an image), but the '`Window`' variable still refers to nowhere. Insert another line into the `Form_Load()` subroutine:

```
Private Sub Form_Load()
    Set Window = HWindowXCtrl1.Object
    Label1.Caption = "Click Next to start"
End Sub
```

   Now the '`Window`' variable refers to the `HWindowX`-part of our ActiveX control.

5. Switch to `Command1`. Another subroutine shows up, which you have to complete like this:

```
Private Sub Command1_Click()
    Call Monkey.ReadImage ("monkey")
    Call Window.DispObj(Monkey)
End Sub
```

   While typing, you will notice a very convenient Visual Basic feature: since it knows the methods of a class, it allows you to select one from a list — if you wish to do so (see figure 3.2). You will also get assistance in supplying the parameter values for a method call in the right order and with the right types (see figure 3.3).

6. Start your application by pressing `F5` and see what happens!

Figure 3.2: *Visual Basic helping you to select a method.*

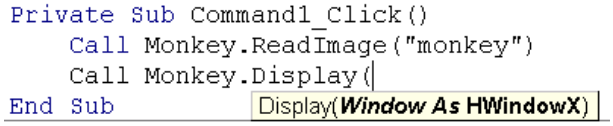

Figure 3.3: *Visual Basic helping you with the correct parameters.*

3.3 Final Step: More Functionality

What we have now is a very basic application which can't do very much — but it needs only 10 lines of code! Thus we will extend the functionality, turning our application into a small image processing demo:

1. Extend the variable declaration section at the beginning of your listing so it looks like this:

```
Dim Monkey As New HImageX
Dim Window As HWindowX
Dim Region As HRegionX
Dim Eyes As HRegionX
Dim State As Integer
```

   Although these declarations are not necesary (Basic declares variables automatically) it is nevertheless a good idea to do so.

2. Check the `Command1_Click()` subroutine and modify it like this:

```
Private Sub Command1_Click()
If State = 3 Then
    End
End If

If State = 2 Then
    Set Eyes = Region.SelectShape("area", "and", 500, 50000)
    Set Eyes = Eyes.SelectShape("anisometry", "and", 1, 1.7)
    Call Window.DispObj(Monkey)
    Call Window.DispObj(Eyes)
    Label1.Caption = "click Finish to terminate"
    Command1.Caption = "Finish"
    State = 3
End If

If State = 1 Then
    Set Region = Monkey.Threshold(128, 256)
    Set Region = Region.Connection()
    Call Window.SetColored(12)
    Call Window.DispObj(Region)
    Label1.Caption = "Next, the ape's eyes will be selected"
    State = 2
End If

If State = 0 Then
    Call Monkey.ReadImage("monkey")
    Call Window.DispObj(Monkey)
    Label1.Caption = "Next, the image will be segmented into
                      several regions"
    State = 1
End If

End Sub
```

3. Run your little program and enjoy a guided tour through a very common image processing example.

3.4 Other Examples

There are some more pre-coded examples, so you can discover how things work with HALCON/COM. These examples can be found under the following directory:

```
%HALCONROOT%\examples\vb
```

The following list shows all the supplied examples and explains their topics in short. To experiment with these examples we recommend to create a private copy in your working directory.

1. `%HALCONROOT%\examples\vb\Applications\FA\`
   An example showing how to use correlation-based pattern matching.

2. %HALCONROOT%\examples\vb\Applications\Monitoring\
   An example showing how to use a background estimator for traffic monitoring.
3. %HALCONROOT%\examples\vb\Online\Barcode\
   An online example showing how to read barcode.
4. %HALCONROOT%\examples\vb\Online\Measure\
   An interactive example showing how to use the measure tool.
5. %HALCONROOT%\examples\vb\Online\Movement\
   An example showing how to discover movement by using the difference of images.
6. %HALCONROOT%\examples\vb\Segmentation\
   An example illustrating the possibilities for interactive image processing applications.
7. %HALCONROOT%\examples\vb\Tools\Calibration\
   An example showing how to calibrate a camera.
8. %HALCONROOT%\examples\vb\Tools\Matching\
   An example showing how to use shape-based matching.
9. %HALCONROOT%\examples\vb\Tools\Measure\
   A “real world” measurement example showing how to measure the pins of an IC.
10. %HALCONROOT%\examples\vb\Manual\
    The example described in this chapter.

Bibliography

[Arm98] Tom Armstrong. *Active Template Library – A Developer's Guide*. M&T Books, Foster City, CA, 1998.

[Bro95] Kraig Brockschmidt. *Inside OLE*. Microsoft Press, Redmond, Washington, 1995.

Index

abstract class, 5–8
ActiveX, 1
ActiveX container, 7
ActiveX control, 7
aggregation, 2, 7

base class, 5, 7, 8
binary reuseability, 1
binding, 3
`BSTR`, 12

`Call`, 13
`Cast()`, 7–9
class definition, 2
classes, 5, 6
code generation, 8
`CommandButton`, 18
component, 1
constructor, 9, 11
containment, 2

data members, 11
data type, 6, 7
default interface, 11
Delphi, 1
derived class, 7, 11
`Dim`, 13
DirectX, 1
`double`, 12

early binding, 3
encapsulation, 2
error handling, 12
error offset, 12
`ErrorBaseCOM`, 12
`ErrorBaseHalcon`, 12
example, 17

form, 17

get properties, 11
group class, 6
GUI, 17

HALCON window, 7
`HalconX`, 5
HDevelop, 7
`HImageX`, 20
`HImageX`, 7, 8
`HWindowX`, 20
`HObjectX`, 7
`HOperatorSetX`, 7, 8
`HRegionX`, 7
`HSystemX`, 12
`HTuple`, 12
`HUntypedObjectX`, 7, 8
`HWindowX`, 7
`HWindowXCtrl`, 7

`IDispatch`, 2
`IHImageX`, 11
`IHObjectX`, 11
implementation, 2
inheritance, 2, 5, 11
inheritance simulation, 5
initialization, 8
instantiation, 8
interface, 2
interfaces, 5, 11
internal state, 6
`IUnknown`, 2

Java, 2

`Label`, 18
language independence, 1
late binding, 3
library, 5
location transparency, 1
`long`, 12

meaning of objects, 6
member, 2
memory management, 9
method, 2
methods, 5, 11

naming convention, 5
`New`, 14, 20
`Nothing`, 14

object, 2
object browser, 17
object copying, 9
object destruction, 9
object duplication, 9
object initialization, 20
object instantiation, 20
object orientation, 1
OLE, 1
`On Error`, 14

pointer, 8, 11
polymorphism, 2
properties, 5, 11
property, 2
put properties, 11

reference, 11
`Resume Next`, 15

semantics of objects, 6
`Set`, 14
special classes, 7
strong typed, 7

top-level window, 7
type conversion, 8

variable, 9
variable declaration, 20, 21
`VARIANT`, 12
versioning, 1
Visual Basic, 2, 13
VisualC++, 1

weak typed, 7